Efficient Exact Two-Level Hazard-Free Logic Minimization

Chris Myers*
Department of Electrical Engineering
University of Utah
myers@ee.utah.edu

Hans Jacobson†
Department of Computer Science
University of Utah
hans@cs.utah.edu

Abstract

This paper presents a new approach to two-level hazard-free sum-of-products logic minimization. No currently available minimizers for single-output literal-exact two-level hazard-free logic minimization can handle large circuits without synthesis times ranging up over thousands of seconds. The logic minimization approach presented in this paper is based on state graph exploration in conjunction with single-cube cover algorithms. Our algorithm achieves fast logic minimization by using compacted state graphs and cover tables and an efficient algorithm for single-output minimization. Our exact two-level hazard-free logic minimizer finds minimal number of literal solutions and is significantly faster than existing literal exact methods — over two orders of magnitude faster for the largest extended burst-mode benchmarks to date. This includes a benchmark that has never been possible to solve exactly in number of literals before.

1 Introduction

In recent years, there has been a number of successful and practical asynchronous circuits designed using asynchronous finite state machines in the form of extended burst-mode controllers [3, 19, 15, 23, 7]. During the design process, it is necessary to try different protocols and state assignments to find a good circuit implementation. This often leads to many iterations through the synthesis process. The bottleneck of finite state machine synthesis is typically in the two-level hazard-free logic minimization step. Since existing exact synthesis methods are too slow, and may not even complete for large circuits, designers are often forced to use heuristic methods, or partitioning, to interactively explore the design space in a reasonable amount of time often yielding sub-optimal circuits. This paper describes a new exact single-output two-level hazard-free logic minimization method that utilizes fast algorithms based on state graph exploration to produce solutions for extended burst-mode controllers in a fraction of the time of the state-of-the-art tools. This new method yields literal exact solutions even for a controller that previously could not be solved exactly in number of literals.

The state-of-the-art in literal exact two-level hazard-free logic minimization is `HFMIN` [4]. However, this tool requires thousands of seconds for large controllers, and does not complete for the largest benchmarks. The tool `IMPYMIN` [21], which is based on implicit algorithms, performs substantially better but is only capable of producing cube-exact solutions. The most time efficient tool to date is `ESPRESSO-HF` [21], but it uses heuristic algorithms and thus cannot guarantee either literal or cube exact solutions. These minimizers are currently used to perform logic minimization in the `3D` [22] and `MINIMALIST` [5] burst-mode synthesis tools.

The approach taken in this paper exploits standard synthesis techniques traditionally used for gate-level speed-independent synthesis, namely state graph exploration and derivation of single-cube covers [16, 1, 2, 11]. In previous work, we successfully applied these algorithms to the synthesis of *extended burst-mode state machines* implemented using *generalized C-elements* (gC) [8]. We not only achieved two to three orders of magnitude improvement in runtime compared with the best literal-exact tool, `HFMIN`, but our tool is also more than one order of magnitude faster than the best heuristic tool, `ESPRESSO-HF`. To be able to achieve these results, we developed a new approach that allows state graphs to be represented very compactly, transforming the graph traversal complexity from potentially exponential, to linear. Compacted states also allow reduced synthesis time in the subsequent binate and unate cover problems by generating smaller cover tables. This paper also develops a new algorithm to efficiently derive single-output covers that are minimal in the number of literals. By dividing the cover problem into more easily solved sub-problems and then merging the results, the time spent by the algorithm in finding a minimal solution is significantly reduced.

The limitation of our previous work is that the circuit implementations required specialized gCs to be in the gate library. Such cells are not typically found in standard-cell or gate-array libraries, and may need to be specially designed. In order to facilitate efficient semi-custom design of

*Supported in part by NSF CAREER award MIP-9625014, SRC contract 97-TJ-487, and a grant from Intel Corporation.

†Supported in part by the University of Utah Graduate Research Fellowship award, and NSF Awards MIP-9622587 and MIP-9988329.

1522-8681/01 $10.00 © 2001 IEEE

integrated circuits, it is necessary to synthesize to standard gates such as ANDs and ORs. The key difficulty in doing so for asynchronous controllers is the need to avoid introducing hazards. In restricting our library to compact, atomic gCs, most hazard concerns are avoided. In particular, the only hazard issue that needs to be addressed is for *dynamic* $0 \rightarrow 1$ transitions. However, when targeting standard gates, additional hazards for *static* $1 \rightarrow 1$ and *dynamic* $1 \rightarrow 0$ transitions may also occur.

This paper leverages our previous work to address the important milestone of achieving fast and exact hazard-free two-level logic minimization. In order to allow the use of standard gate libraries, our synthesis methodology is extended to guarantee the absence of all static 1 hazards and dynamic hazards for $1 \rightarrow 0$ transitions as well as dynamic $0 \rightarrow 1$ transitions. By combating state explosion through compacted state graphs and dividing the minimization problem into more easily solved sub-problems, our new algorithm allows very fast logic minimization of even the largest benchmarks to date and yields per-output literal-exact solutions, better supporting interactive and iterative exact exploration of design alternatives.

Section 2 gives a background on asynchronous logic synthesis and the hazard models used in two-level logic minimization. Section 3 gives background on compacted state graphs used by our new method. Section 4 presents a new exact algorithm that achieves per-output literal-exact hazard-free covers using standard gates along with a detailed example. Section 5 presents benchmark comparisons between the algorithm presented in this paper and other available minimization tools. Conclusions can be found in Section 6.

2 Asynchronous logic synthesis background

For asynchronous design, the two-level logic minimization problem is complicated by the fact that there must be no hazards in the sum-of-products (SOP) implementation. This section describes our assumptions and hazard model.

2.1 Extended burst-mode machines

The synthesis method described in this paper produces hazard-free implementations for controllers specified as *extended burst-mode* (XBM) machines, a class of multiple input change (MIC) asynchronous finite state machines. Inputs are allowed to change concurrently in a form of MIC called *bursts*. The signals within a burst may arrive in arbitrary order. Every *input burst* if followed by a (possibly empty) concurrent burst of output and state signals changes. After the output and state burst, the internal nodes of the circuit are allowed to stabilize before the fed back state signals are allowed to reach the inputs of the circuit. The circuit operates under *fundamental-mode* which means that the circuit must be allowed to stabilize in response to the fed back state signals before the next input burst can arrive.

The extended burst-mode specification shown in Figure 1 is used as an example in later sections. Each transition

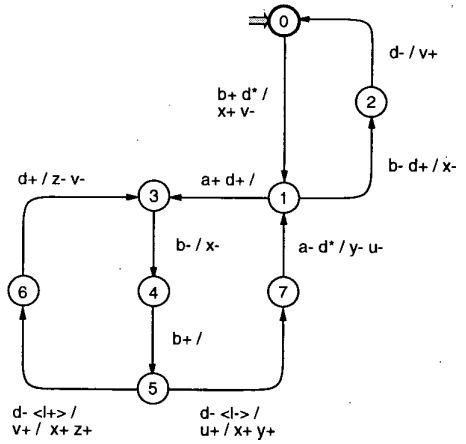

Figure 1. Example: extended burst-mode controller *ack-xbm-si*.

between states contains an input burst and a possibly empty output burst. If there is more than one outgoing transition from a state, a deterministic choice is implied. Signals annotated with $+$ and $-$ signs and not enclosed in angle brackets imply a rising and falling transition, also known as a *terminating edge*. Input signals annotated with a $*$ are called *directed don't cares* and are free to change monotonically at any time during a sequence of specified directed don't care states but must change by the time the signal is next specified as a terminating edge. A terminating edge not preceded by a directed don't care is called a *compulsory edge*. Signals enclosed within angle brackets are *conditionals* (level signals) and are free to change non-monotonically whenever not specified. When specified, such a conditional is assumed to reach a stable value in time to be sampled correctly by the arriving compulsory edges of the burst. State transitions occur only when all conditionals are met and all terminating edges pertaining to a burst have appeared.

Transitions in XBM machines begin in one cube c_1 and end in another cube c_2 where the values of multiple variables may change during the transition. The cube c_1 is called the *start cube* and c_2 is called the *end cube* of the transition. The smallest cube that contains both c_1 and c_2 is called the *generalized transition cube* and is denoted $[c_1, c_2]$ [25]. This cube includes all possible minterms that a machine may pass through starting in c_1 and ending in c_2. A generalized transition cube can also be represented with a product which contains a literal for each variable x_i in which $c_1(i) = c_2(i) \neq -$. An *open generalized transition cube* $[c_1, c_2)$ includes all minterms in $[c_1, c_2]$ except those in c_2. An open transition cube usually must be represented using a set of products.

In a generalized transition cube, signals may be of type rising, falling, or level. Rising and falling signals change monotonically (i.e., at most once in a legal generalized transition cube). Level signals must hold the same value in c_1

and c_2, where the value is either a constant (0 or 1) or a don't care (−). Level signals, if they are don't care, may change non-monotonically.

2.2 Hazard model

If a function f does not change monotonically during a MIC, then f has a *function hazard* for that transition. If a transition has a function hazard, there exists no implementation of the function which avoids the hazard during the transition. In an XBM machine, the types of transitions are restricted such that each function may only change value after the completion of an input burst. A generalized transition $[c_1, c_2]$ for a function f is an *extended burst-mode transition* if for every minterm $m_i \in [c_1, c_2)$, $f(m_i) = f(c_1)$ and for every minterm $m_i \in c_2$, $f(m_i) = f(c_2)$. Therefore, if a function only has extended burst-mode transitions, then it is function hazard-free [17, 25].

Although there are no functions hazards, there may be *logic hazards*. In order to design a hazard-free SOP cover, we must consider each possible type of transition in turn. First, if during a static $1 \rightarrow 1$ transition $[c_1, c_2]$, the cover of f can momentarily evaluate to 0, then there exists a static 1-hazard. In a SOP cover, consider the case where there is one product p_1 which contains c_1 but not c_2 and another product p_2 which contains c_2 but not c_1. The cover includes both c_1 and c_2, but there can be a static 1-hazard. If p_1 is implemented with a faster gate than p_2, then the gate for p_1 may turn off faster than the gate for p_2 turns on which can lead to the cover momentarily evaluating to a 0, thus exhibiting a momentary $1 \rightarrow 0 \rightarrow 1$ glitch on the output. When multiple variables are changing concurrently, the cover may pass through other minterms along the way between c_1 and c_2. To be free of static 1-hazards, it is necessary that a single product in the cover include all these minterms. In other words, each generalized transition cube, $[c_1, c_2]$, where $f(c_1) = f(c_2) = 1$, must be contained in some product in the cover to eliminate static 1-hazards.

In a SOP cover of a function, the only way there can be a static 0-hazard is if some product includes both signal, x_i, and its complement $\overline{x_i}$. Clearly, such a product would not be included in a minimal SOP cover, so SOP covers never produce a static 0-hazard.

The results for dynamic hazards are a little more complicated, and we need a couple more definitions. The *start subcube*, c_1', is a maximal subcube of c_1 such that each signal undergoing a directed don't care transition is set to its initial value (i.e., 0 for a rising transition and 1 for a falling transition). The *end subcube*, c_2', is a maximal subcube of c_2 such that each signal undergoing a directed don't care transition is set to its final value (i.e., 1 for a rising transition and 0 for a falling transition).

For each dynamic $1 \rightarrow 0$ transition, $[c_1, c_2]$, if a product in the SOP cover intersects $[c_1, c_2]$ (i.e., it includes a minterm from the transition), then it must also include the start subcube, c_1'. For each dynamic $0 \rightarrow 1$ transition, $[c_1, c_2]$, if a product in the SOP cover intersects $[c_1, c_2]$, then it must also include the end subcube, c_2'. If a product term contains c_1' or c_2', then it changes value monotonically during the transition. If a product term does not contain c_1' or c_2' but still intersects the transition cube $[c_1, c_2]$, then it may change value non-monotonically during the transition. During the transition from c_1 to c_2, the product term starts out as 0, then evaluates to 1 when the transition crosses the intersection point, and then goes back to 0 once the transition has passed the intersection point. For a dynamic $1 \rightarrow 0$ transition this could result in a $1 \rightarrow 0 \rightarrow 1 \rightarrow 0$ glitch if the intersecting product term is slow to turn on. Similarly, a dynamic $0 \rightarrow 1$ transition could result in a $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$ glitch if the intersecting product term turns on and off quickly [25].

2.3 Summary of hazard requirements

The hazard-free cover requirements for the ON-set of a two-level SOP implementation of output function f are:

1. Each ON-set cover cube of f must not intersect the OFF-set.
2. Every static $1 \rightarrow 1$ transition $[c_1, c_2]$ in f must be completely covered by some product.
3. For every dynamic $0 \rightarrow 1$ transition $[c_1, c_2]$ in f, c_2 must be completely covered by some product.
4. For every dynamic $1 \rightarrow 0$ transition $[c_1, c_2]$ in f, each maximal subcube of $[c_1, c_2)$ must be completely covered by some product.
5. Any product term of f intersecting a dynamic $1 \rightarrow 0$ transition $[c_1, c_2]$ must also contain the start subcube c_1'.
6. Any product term of f intersecting a dynamic $0 \rightarrow 1$ transition $[c_1, c_2]$ must also contain the end subcube c_2'.

Requirements 2, 3, and 4 describe the product terms, also called *required cubes*, that are required for the cover to turn on when it is supposed to. The entire required cube must be covered by some product in the cover to eliminate static hazards. The transition cubes described in requirements 5 and 6 are also called *privileged cubes*. Intersections of these cubes that do not contain the appropriate subcube result in dynamic hazards. Therefore, a hazard-free cover must guarantee there are no such illegal intersections. As shown in Section 4, all of these requirements are satisfied by the logic minimization algorithm presented in this paper.

3 Compacted state graphs

Our algorithm begins with a *compacted state graph* (CSG)[8]. CSGs is an efficient way to represent states of highly concurrent controllers without introducing state explosion. Similar methods to avoid state explosion have been explored for speed-independent property analysis of gate networks [10]. A CSG is modeled with a 4-tuple $\langle I, O, \Phi, \Gamma \rangle$ where I is a set of input signals, O is a set of output signals,

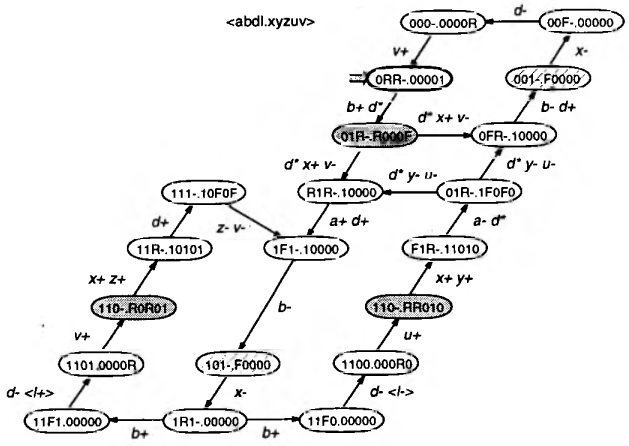

Figure 2. CSG for controller *ack-xbm-si*.

Φ is a set of *compacted states*, and Γ is a set of state transitions. Each compacted state is labeled with a cube consisting of all states reachable by all possible interleavings of the set of currently enabled monotonic (edge) and non-monotonic (level) input and output signals. For a given compacted state s and signal u, we define the function $s(u)$ as follows:

$$s(u) = \begin{cases} 0 & \text{stable low} \\ R & \text{changing monotonically from 0 to 1} \\ 1 & \text{stable high} \\ F & \text{changing monotonically from 1 to 0} \\ - & \text{changing non-monotonically} \end{cases}$$

CSGs can be easily derived from both BM and XBM state machines. CSGs derived from XBM specifications are composed of compacted states which are one of two different types. A compacted state is said to represent an *input burst* if some inputs are changing while all outputs are stable. A compacted state is said to represent an *output burst* if some outputs are unstable. Due to directed don't cares and level signals, some output bursts may contain unstable inputs. In a CSG derived from an XBM machine, the successor state of an input burst must be either a unique output burst or possibly a set of input bursts. Figure 2 illustrates the CSG derived from the *xbm-si* XBM specification in Figure 1.

CSGs can be used to represent the set of generalized transition cubes implementing a function in a compact fashion. A compacted state represents a generalized transition cube for each output. The transition cube for a given compacted state s is $[initial(s), final(s)]$ and the product is $cube(s)$, where *initial*, *final*, and *cube* can be determined using the following three functions:

$$\text{initial}(s)(u) = \begin{cases} 0 & \text{if } s(u) = 0 \text{ or } s(u) = R \\ 1 & \text{if } s(u) = 1 \text{ or } s(u) = F \\ - & \text{if } s(u) = - \end{cases}$$

$$\text{final}(s)(u) = \begin{cases} 0 & \text{if } s(u) = 0 \text{ or } s(u) = F \\ 1 & \text{if } s(u) = 1 \text{ or } s(u) = R \\ - & \text{if } s(u) = - \end{cases}$$

$$\text{cube}(s)(u) = \begin{cases} 0 & \text{if } s(u) = 0 \\ 1 & \text{if } s(u) = 1 \\ - & \text{if } s(u) = - \text{ or } s(u) = R \text{ or } s(u) = F \end{cases}$$

For each output signal u and each input burst s, we can determine what type of transition is represented by the transition cube for the state. Assume that $s(u) = 0$. First, we must check s' where $(s, s') \in \Gamma$. If $s'(u) = 0$, then the transition cube, $cube(s)$, is a $0 \rightarrow 0$ transition. If $s'(u) = R$, then $cube(s)$ is a $0 \rightarrow 1$ transition. Now assume $s(u) = 1$. If $s'(u) = 1$ then $cube(s)$ represents a $1 \rightarrow 1$ transition. Finally, if $s'(u) = F$, then $cube(s)$ is a $1 \rightarrow 0$ transition. For each output burst s, the transition cube, $cube(s)$, represents the state of signals during the feedback of the output variables back to the inputs. Feedback is always a static transition for an output. Thus if $final(s)(u) = 1$, it is a static $1 \rightarrow 1$ transition, and if $final(s)(u) = 0$, it represents a static $0 \rightarrow 0$ transition.

The *maximal subcubes* for a compacted state are the cubes formed by, for each cube, setting one don't care corresponding to a terminating input edge to its initial value. In order to find the maximal subcubes, we split the state for each such don't care signal using the following function:

$$\text{split}(s, v)(u) = \begin{cases} 0 & \text{if } s(u) = 0 \text{ or } (u = v \text{ and } s(u) = R) \\ 1 & \text{if } s(u) = 1 \text{ or } (u = v \text{ and } s(u) = F) \\ - & \text{if } u \neq v \text{ and } (s(u) = - \text{ or } s(u) = R \\ & \text{or } s(u) = F) \end{cases}$$

Table 1 illustrates the results of the respective functions when applied to state s = *abdl.xyzuv:0FR-.10000* of the CSG in Figure 2. In the table, v = $\{a,b,d,l,x,y,z,u,v\}$.

initial(s)	final(s)	cube(s)	split(s,v)
010-.10000	001-.10000	0---.10000	01--.10000
			0-0-.10000

Table 1. State s = *0FR-.10000* applied to defined functions.

4 Hazard-free logic minimization algorithm

Our logic minimization algorithm uses state graph traversal and ideas similar to those in the fast single-cube algorithm [16] designed for speed-independent and timed circuits. An important property of generalized transitions is that a required ON-set cube can always be represented by a single cover cube. Therefore, taking a single-cube approach is guaranteed to always find an optimal solution for two-level implementations. Figure 3 presents our logic minimization algorithm. The input to our logic minimization algorithm is a CSG derived from a state assigned XBM machine, and the output is a minimum literal logic equation for each output signal implementing the ON-set regions in the CSG. The logic minimization algorithm begins by traversing the CSG to derive the set of required cubes necessary to cover the ON-set. The algorithm then proceeds to find the initial *trigger*

cubes for each required cube separately. A trigger cube is an initial cover that includes signals that must appear in any correct cover of the required cube. Since this initial cover may include minterms from the OFF-set, called *cover violations* (CV), as well as dynamic hazards, called *intersection violations* (IV), additional *context signals* may have to be added to the cover. A context signal is any signal that is stable in the required cube and which is not a trigger signal. Next, a binate covering problem is formulated to remove all cover and intersection violations by selecting the minimum number of context signals needed. The result is a set of solutions derived for each required cube. Finally, a unate covering problem is setup and solved to find the minimum number of these solutions needed to cover all required cubes for each output signal.

```
minimize(⟨I,O,Φ,Γ⟩)
  foreach u ∈ O
    LocalSol = ∅;
    RC = find_required_cubes(⟨I,O,Φ,Γ⟩,u);
    foreach rc ∈ RC
      tc = find_trigger_cube(⟨I,O,Φ,Γ⟩,u);
      CS = find_context_signals(I,O,rc,tc);
      CV = find_cover_violations(⟨I,O,Φ,Γ⟩, RC, u, tc);
      IV = find_intersection_violations(⟨I,O,Φ,Γ⟩, CS, u, rc, tc);
      table = build_binate_cover_table(tc, CS,CV, IV);
      LocalSol = LocalSol ∪ solve_binate_cover_table(table,RC);
    table = build_unate_cover_table(RC, LocalSol);
    Sol(u) = solve_unate_cover_table(table);
  return(Sol);
```

Figure 3. Logic Minimization Algorithm.

4.1 Finding required cubes

The set of required cubes describes how the ON-set of the function must be covered in order to obtain a hazard-free solution. Each required cube must be completely covered by some product term. The following required cubes must be generated for each generalized transition $[c_1, c_2]$:

- $0 \rightarrow 1$ transition: required cube equals c_2.
- $1 \rightarrow 1$ transition: required cube equals $[c_1, c_2]$.
- $1 \rightarrow 0$ transition: set of required cubes is the maximal subcubes of $[c_1, c_2)$.
- $0 \rightarrow 0$ transition: no required cube.

Required cubes are derived during a traversal of the CSG as shown in Figure 4. For each compacted state, s, the algorithm checks the value of the output signal, u, that is being synthesized. If the value of u in s is R, then the compacted state is an output burst state representing a $0 \rightarrow 1$ transition. This has a required cube equal to the initial cube of this burst. This state also represents a $1 \rightarrow 1$ transition when the output signals are fed back. This cube is equal to the entire state cube which includes the first required cube. Therefore, only the second required cube is necessary.

```
find_required_cubes(⟨I,O,Φ,Γ⟩,u)
  RC = ∅;
  foreach s ∈ Φ
    if s(u) = R then  // Dynamic 0 → 1 and static 1 → 1 transitions
      RC = RC ∪ cube(s);
    else if s(u) = 1 then
      if ∃(s,s') ∈ Γ . s'(u) = 1 then  // Static 1 → 1 transition
        RC = RC ∪ cube(s);
      else  // Dynamic 1 → 0 transition
        foreach v ∈ I ∪ O
          if (s(v) = R and s'(v) = 1) or
             (s(v) = F and s'(v) = 0) then
            RC = RC ∪ split(s,v);
    else if s(u) = 0 or s(u) = F then ;  // No required cube
  return(RC);
```

Figure 4. Algorithm to find required cubes.

If the value of u in s is 1, this represents either a $1 \rightarrow 1$ transition or a $1 \rightarrow 0$ transition. If it is a $1 \rightarrow 1$ transition then the successor state also has u as 1. In this case, the required cube is the state cube. If the value of the output in the successor state is F, then the required cubes are all the maximal subcubes of s. These can be found by checking each unstable signal for stability in the successor state. If this signal is stabilizing, then a cube is formed where all other unstable signals are don't care, and this signal is stable at its initial value.

If the value of u in s is 0, this represents either a $0 \rightarrow 0$ transition or a $0 \rightarrow 1$ transition. A required cube is only needed if it is a $0 \rightarrow 1$ transition. As mentioned above, this required cube is included in the next compacted state (the output burst state). Therefore, it does not need to be generated for this compacted state.

If the value of u is F, then the compacted state must be an output burst state representing a $1 \rightarrow 0$ transition. The required cubes for this transition are found when the preceding compacted state (the input burst) is processed as described above.

Consider the CSG in Figure 2 as an example. The notation *x+,3* corresponds to the third occurrence of signal x rising in the CSG in a depth first search (left branch first) starting in the start state *abdl.xyzuv:0RR-.00001*. Similarly, *x*,1* refers to the first static transition of x, and *x-,1* to the first falling transition of x. Consider the output signal x. The logic minimization starts out by first deriving the required cubes for x. Take required cube *RC(x+,2) = 110-.-0-01* as an example. This required cube corresponds to the leftmost shaded compacted state, *S = 110-.R0R01*, in Figure 2 where x is enabled to rise. This required cube covers the static $1 \rightarrow 1$ transition for x caused by the output variable feedback in state S. The required cubes for x are illustrated in Table 2.

4.2 Finding trigger cubes and context signals

Each required cube has an associated trigger cube which forms the initial cover for that required cube. While a final hazard-free cover cube can always be found from an empty trigger cube, adding initial trigger signals narrows the search for the cover cube. These trigger signals are formed by signals known to be required for a legal cover and are derived from the current transition.

The transition cube of a $0 \rightarrow 1$ transition $[c_1, c_2]$ is restricted to turn on only once the end cube c_2 has been reached. c_2 is reached only once all terminating signals in the transition cube have fired. The final cover cube for this transition is therefore required to contain the value of all terminating edge signals after they have fired — otherwise the final cover cube would intersect the OFF-set. The terminating edge signals of the transition are therefore added to the trigger cube as initial trigger signals.

Similarly, each trigger cube for a $1 \rightarrow 0$ transition $[c_1, c_2]$ is required to turn off as the end cube c_2 is entered. Each required cube, being a maximal subcube of $[c_1, c_2)$, must therefore turn off as the last signal in the trajectory it covers fires. Each trigger cube can therefore be annotated with the initial value of the last signal to fire in its corresponding required cube.

Note that a signal specified as a directed don't care in a transition cannot be a trigger signal since that transition has its value as a don't care throughout the transition cube. A level signal is also not a trigger signal since the setup time requirement forces it to stabilize before any compulsory edges arrive at the inputs. Once a trigger cube is found, the context signals are those signals that are not trigger signals and are stable in the required cube.

In our example, after generating the required cubes, the algorithm proceeds to find the context signals and trigger cubes corresponding to these required cubes. For the required cube *RC(x+,2)* derived from state *110-.R0R01* in our example, the corresponding trigger cube is *TC(x+,2)* = `----.----1` since *v+* is the signal transition that causes the machine to enter the required cube and turn on the cover. The corresponding context signals are *a, b, d', y', u'* since these signals are stable throughout the required cube. The trigger cubes and context signals for *x* are illustrated in Table 2.

4.3 Finding violations

Since the initial trigger cube may span over minterms belonging to the OFF-set (i.e., a *cover violation*), and may also intersect other transition cubes in a hazardous way (i.e., an *intersection violation*), such violations must be removed.

A trigger cube *tc* for a signal *u* contains a cover violation if it intersects a transition $[c_1, c_2]$ in which the value of *u* is 0. The algorithm shown in Figure 5 finds all cover violations. In a compacted state *s*, signal *u* is or is tending to 0 when it has value 0 or *F*, and if *tc* intersects *s*, then there may be a cover violation. There is, however, one special case. If the value

Γ	Required cubes	Trigger cubes	Context signals
x+,1	`01--.-000-`	`-1--.-----`	a' y' z' u'
x*,1	`-1--.10000`	`----.-----`	b x y' z' u' v'
x-,1	`111-.10000`	`-1--.-----`	a d x y' z' u' v'
x+,2	`110-.-0-01`	`----.----1`	a b d' y' u'
x*,2	`11--.10101`	`----.-----`	a b x y' z u' v
x*,3	`111-.10-0-`	`----.-----`	a b d x y' u'
x+,3	`110-.--010`	`----.---1-`	a b d' z' v'
x*,4	`-1--.11010`	`----.-----`	b x y z' u v'
x*,5	`01--.1-0-0`	`----.-----`	a' b x z' v'
x-,2b	`01--.10000`	`-1--.-----`	a' x y' z' u' v'
x-,2d	`0-0-.10000`	`--0-.-----`	a' x y' z' u' v'

Table 2. Required cubes, trigger cubes, and context signals for the transitions (Γ) of output *x* of controller *ack-xbm-si*.

of *u* is 0 in an input burst and *R* in the following output burst then all states in the compacted state except the final state (where *u* is tending to 1) must be excluded. Therefore, in this case all maximal subcubes must be found and excluded individually (if they intersect the trigger cube).

```
find_cover_violations(⟨I,O,Φ,Γ⟩,RC,u,tc)
  CV = ∅;
  foreach s ∈ Φ
    if (s(u) = 0 or s(u) = F) and tc ∩ s ≠ ∅ then
      if ∃(s,s') ∈ Γ . s'(u) = R then
        foreach v ∈ I ∪ O
          if ((s(v) = R and s'(v) = 1) or
              (s(v) = F and s'(v) = 0)) and
              (split(s,v) ∩ tc ≠ ∅) then
            CV = CV ∪ split(s,v);
      else
        CV = CV ∪ s;
  return(CV);
```

Figure 5. Algorithm to find cover violations.

A trigger cube *tc* contains an intersection violation (also called illegal intersection) if *tc* intersects a dynamic $1 \rightarrow 0$ transition $[c_1, c_2)$ and *tc* does not contain the start subcube c_1'. An intersection violation also exists if *tc* intersects the end cube, c_2 for a dynamic $0 \rightarrow 1$ transition and does not contain the end subcube c_2'. Intersection violations can also be caused by the selection of a context signal that, when added to the trigger cube, causes an illegal intersection of a dynamic transition.

Intersection violations can be detected through a state graph traversal as shown in the algorithm in Figure 6. This algorithm returns a set of intersection violations in which each violation is a pair where the first element is the choice of context signal that causes the violations and the second is the state which must be excluded.

For each dynamic $1 \rightarrow 0$ transition state *s*, the algorithm checks if the trigger cube, *tc*, intersects *s*, but does not contain the start subcube. The start subcube, c_1', is found by setting all changing signals to their initial value. Even if *tc* contains c_1', the choice of a context signal may lead to an illegal intersection, and a context signal must be added to re-

```
find_intersection_violations(⟨I,O,Φ,Γ⟩,CS,u,rc,tc)
  IV = ∅;
  foreach s ∈ Φ
    // Check if state is dynamic 1 → 0 transition intersecting trigger cube
    if s(u) = 1 and ∃(s,s') ∈ Γ . s'(u) = F and tc ∩ s ≠ ∅ then
      c'1 = initial(s);  // Find start subcube
      if c'1 ⊈ tc then
        IV = IV ∪ { (∅,s) };
      else
        foreach v ∈ CS
          // Check if context signal excludes part of start subcube
          // but not the state
          if rc(v) ≠ c'1(v) and cube(s)(v) = − then
            IV = IV ∪ { (v,s) };
    // Check if state is dynamic 0 → 1 transition intersecting trigger cube
    else if s(u) = R and tc ∩ s ≠ ∅ then
      // Continue if there are level signals or directed don't cares
      if ∃v ∈ I . cube(s)(v) = − then
        c'2 = s;  // Copy for end subcube
        foreach v ∈ O
          s'(v) = initial(s)(v);
        c'2 =final(s');  // Find end subcube
        if tc ∩ s' then
          if c'2 ⊈ tc then
            IV = IV ∪ { (∅,s'); }
          else
            foreach v ∈ CS
              // Check if context signal excludes part of end
              // subcube but not the state
              if rc(v) ≠ c'2(v) and cube(s')(v) = − then
                IV = IV ∪ { (v,s'); }
  return(IV);
```

Figure 6. Algorithm to find intersection violations.

move it. In other words, if a choice of a context signal causes the cover to exclude part of the start subcube but still intersect the state, then we must choose another context signal that excludes the whole state s.

For each dynamic $0 \rightarrow 1$ transition, if there are no unstable input signals or conditional signals, then the end cube equals the end subcube and is a single minterm so there can be no violation (i.e., this is a burst-mode transition). If this is not the case, we set all outputs in the state to their initial value since the $0 \rightarrow 1$ transition takes place before the changes in output values are allowed to be fed back. To calculate the end subcube, we set all unstable input signals to their final value. If the trigger cube intersects this compacted state, but does not include the end subcube, then there is an intersection violation and a context signal must be added to remove this compacted state. If a context signal choice can exclude part of the end subcube while still allowing the cover to intersect the state, then another context signal must be chosen to remove the rest of the state.

Consider trigger cube *TC(x+,2) = ----.----1* in our example. This trigger cube has a cover violation since it intersects compacted state *0RR-.00001* which belongs to the OFF-set of x. Since x goes high once b rises, the cover violation is represented by cube *00R-.00001*. The trigger cube also has a potential intersection violation of state *0RR-.00001*. If *d'* was to be selected as a context signal, the trigger cube would no longer cover the end subcube *011-.00001* of the transition and the cover would be hazardous. If this context signal is added to the cover (to remove other violations), the entire end cube *01R-.00001* of state *0RR-.00001* must be excluded in order to remove the introduced hazard. This is achieved by selecting other context signals that exclude the entire end cube from the cover. The cover and intersection violations for output x are shown in Table 3.

TC	CV	IV
x+,1	1101.00000,11F1.00000 1100.00000,11F0.00000	none
x*,1	1R1-.00000,1101.00000 11F1.00000,1100.00000 11F0.00000,101-.F0000 00R-.00001,001-.F0000 000-.0000R,00F-.00000	none
x-,1	CV(x+,1)	0FR-.10000
x+,2	00R-.00001	01R-.00001
x*,2	CV(x*,1)	none
x*,3	CV(x*,1)	0FR-.10000
x+,3	none	none
x*,4	CV(x*,1)	none
x*,5	CV(x*,1)	none
x-,2b	CV(x+,1)	none
x-,2d	1101.00000,1100.00000 00R-.00001,000-.0000R	01R-.00001

Table 3. Cover violations (CV) and intersection violations (IV) for the trigger cubes (TC) of output x of controller *ack-xbm-si*.

4.4 The covering tables

In order to find the best choice of context signals to remove all violations, our logic minimization algorithm sets up and solves a covering problem where the columns of the covering table are the available context signals and the rows are the violations that must be removed. Since selecting certain context signals can lead to more intersection violations, this covering problem is binate. A row is added to the binate cover table for every cover violation along with an "X" in the column for every context signal that completely removes the violating state cube. Similarly, a row is added to the binate table for every intersection violation. An "X" is added for every context signal that removes the violation. In addition an "O" is added if the intersection violation is caused by that context signal.

The binate cover table is then solved using classic reduction techniques and a branch and bound algorithm. During column dominance, a dominated context signal is only removed if the dominating context signal excludes less required cubes. Since our ultimate goal is to find the minimum cover of all required cubes, this step is necessary to ensure that a minimum literal solution can be found. During the branch and bound, the algorithm records the set of minimal unique hazard-free cubes that can cover the required cube.

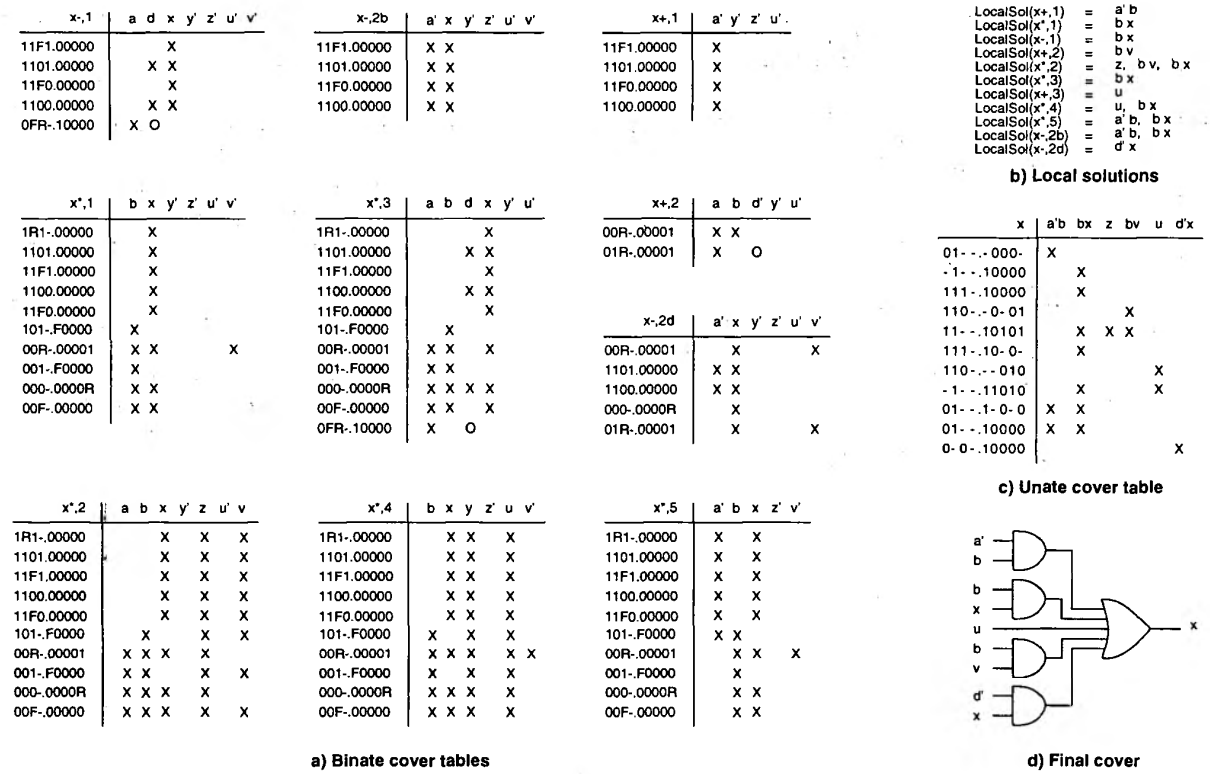

Figure 7. a) binate cover tables, b) local solutions, c) unate cover table, and d) final solution for output x of controller *ack-xbm-si*.

A minimal unique cube is a cube covering a set of required cubes not covered by any other cube of smaller cardinality.

Once a set of hazard-free minimal unique cubes have been derived for all required cubes of an output function, finding the minimal cover is posed as a unate covering problem. In the unate table, the rows represent the required cubes of the function and the columns represent the sets of minimal unique cubes. Solving the unate cover table using classic techniques then results in a final hazard-free cover that is minimal in number of literals.

In our example, consider the binate cover table built for trigger cube $TC(x+,2) = ----.----1$. The context signals *a, b, d', y', u'* of the trigger cube are entered as columns in the table and the two violation cubes *00R-.00001*, and *01R-.00001* are entered as rows. An "O" is entered for row *01R-.00001* in the column for *d'* since adding that context signal will introduce a hazard and require us to remove the row from the table by selecting another context signal. For each row an "X" is entered in each column where the context signal would remove the violation cube. In our case, signals *a* and *b* would remove cover violation *00R-.00001*, and *a* would remove intersection violation *01R-.00001*. In this fashion, binate cover tables are built for each required cube for *x* as illustrated in Figure 7(a).

The binate cover tables are then solved one by one. For trigger cube *x+,2* there exists two minimal unique solution cubes, *bv* and *av*. Since they both cover the same set of required cubes with the same number of literals, only one cube needs to be kept. In our case we keep *bv*. Generally, a trigger cube can have several local solutions. *x*,2* for example has three different local solutions, *z*, *bv*, and *bx*. These are all minimal unique solutions that cover a set of required cubes not covered by any of the other local solutions. For example, while *z* is a smaller solution than *bv*, it covers only required cube *11--.10101* while *bv* covers *11--.10101* and *110-.-0-01*. Thus *bv* is a minimal solution for the unique set of required cubes it covers. Similarly, while *z* covers only a subset of the required cubes that *bv* covers, *z* is a smaller cover, so *z* is still a minimal cover for the set of required cubes {*11--.10101*}. The local solutions for each trigger cube are illustrated in Figure 7(b).

Finally, a unate cover table (see Figure 7(c)) is built with the required cubes as rows, and the combined local solutions as columns. A final minimal literal solution covering all required cubes is then found using classic reduction techniques. In our case, local solution cubes *ab'*, *bx*, *bv*, *u*, and *d'x* are all essential and form the final cover illustrated in Figure 7(d).

			ATACS (literal-exact)		HFMIN (literal-exact)		IMPYMIN (cube-exact)		ESPRESSO-HF (heuristic)	
Design	|Φ|	IO	Time	Lit	Time	Lit	Time	Lit	Time	Lit
ack-cdplayer-p1	88	48	80.36	708	impossible		1254.57	743	85.46	736
ack-fibonacci	60	38	2.98	415	3132.84	415	346.50	522	19.27	494
ack-diffeq	36	35	1.16	261	142.56	261	57.84	287	10.81	288
ack-barcode	38	31	1.35	229	105.50	229	65.27	254	9.02	253
ack-gcd	33	21	0.52	98	18.74	98	23.37	98	4.04	98
ack-factorial	23	17	0.39	40	11.57	40	10.23	40	4.59	40
ack-xbm-si	20	9	0.31	34	6.47	34	9.03	34	2.28	34
cache-ctrl	98	36	238.48	500	1316.79	500	236.59	550	18.58	536
chu-ad-opt-e	8	6	0.29	11	3.62	11	3.86	11	2.19	11
dme-e	20	8	0.30	22	6.30	22	5.58	22	2.13	22
dme-fast-e	20	8	0.31	30	6.06	30	5.68	30	2.18	30
dram-ctrl	26	14	0.47	50	8.93	50	8.28	50	3.10	50
hp-ir-sc-ctrl	76	30	3.11	264	77.47	264	41.57	280	9.99	298
hp-ir-sd-ctrl	54	24	0.95	131	29.91	131	27.62	134	7.64	132
hp-ir-it-ctrl	24	13	0.36	52	10.48	52	9.03	53	3.00	54
hp-ir-rf-ctrl	26	13	0.38	32	8.69	32	7.37	32	3.34	32
hp-ir-two-tick	18	8	0.30	15	4.83	15	4.19	15	2.63	15
hp-ir	16	5	0.28	8	3.21	8	3.20	8	1.56	8
postoffice-pesnd	28	10	0.40	65	5.87	65	6.12	65	2.22	65
postoffice-bufsnd	17	8	0.34	34	5.54	34	5.64	34	2.15	34
postoffice-bufrd	14	7	0.29	15	4.78	15	4.92	15	1.35	15
pscsi-pscsi	119	22	9.63	400	53.42	400	42.40	421	8.33	428
pscsi-isend	22	11	0.39	74	8.19	74	8.08	74	3.28	75
pscsi-ircv	13	9	0.35	31	6.70	31	5.60	31	2.49	31
pscsi-trcv-bm	18	10	0.36	40	8.18	40	7.21	40	3.87	40
pscsi-trcv	14	8	0.34	25	4.73	25	5.12	25	2.71	25
pscsi-tsend-bm	24	11	0.39	62	8.73	62	7.60	62	3.41	62
pscsi-tsend	24	11	0.39	55	8.51	55	7.74	55	3.41	56
sscsi-isend-bm	24	11	0.40	61	7.92	61	7.22	61	3.88	61
sscsi-isend-csm	18	11	0.39	44	6.88	44	6.09	44	3.80	44
sscsi-trcv-bm	24	11	0.39	54	7.34	54	6.54	54	3.85	54
sscsi-trcv-csm	18	11	0.38	43	7.89	43	6.05	43	2.74	43
sscsi-tsend-bm	26	11	0.40	64	7.50	64	8.14	64	3.80	64
sscsi-tsend-csm	22	11	0.39	37	7.32	37	7.01	38	2.88	39
stetson-p1	84	30	4.06	284	78.32	284	42.95	298	9.56	315
stetson-p2	56	24	0.96	148	33.58	148	28.17	153	8.73	155
stetson-p3	18	7	0.30	11	3.33	11	3.78	11	2.23	11
vanbek-ad-opt-e	6	6	0.29	14	3.27	14	3.79	14	1.85	14
xscsi-fifo2scsi	26	11	0.38	83	7.17	83	8.50	84	2.97	83
xscsi-dma2fifo	22	9	0.33	52	6.35	52	7.50	52	2.96	52
xscsi-fifo2dma	16	8	0.35	29	5.07	29	4.61	29	2.65	29
xscsi-fifocellctl	6	5	0.29	11	3.47	11	3.17	11	1.87	11
yun-diffeq-alu2	36	15	0.51	143	14.43	143	15.77	156	5.34	146
yun-diffeq-alu1	18	10	0.33	43	8.65	43	8.18	43	3.38	44
yun-diffeq-mul1	8	7	0.29	42	4.62	42	4.61	42	2.44	42
yun-diffeq-mul2	6	6	0.29	15	3.17	15	3.95	15	1.94	15

Table 4. Benchmark comparisons between existing two-level logic minimizers ATACS (the method presented in this paper), HFMIN, IMPYMIN, and ESPRESSO-HF. (|Φ| - number of compacted states in state graph, *IO* - number of input, output, and state signals, *Time* - logic minimization run-time in seconds, *Lit* - number of literals in solution.)

5 Results

The algorithm described in this paper has been completely incorporated and automated in the ATACS [16] synthesis tool. The ATACS extended burst-mode logic minimizer is exact in number of literals. We compare against the publicly available state of the art hazard-free logic minimization tools developed by Nowick et al. The HFMIN [4] minimizer is exact in number of literals. The IMPYMIN [4] implicit minimizer is exact in number of cubes, but does not perform literal minimization. The ESPRESSO-HF minimizer is heuristic. It should be noted that the HFMIN, IMPYMIN, and ESPRESSO-HF tools are optimized for the more difficult problem of multi-output minimization. As our goal is to generate controllers with small delay rather than small area, the minimizers are run in single-output configurations for the benchmarks presented in this paper.

The runtime and literal comparisons shown in Table 4 contains the largest benchmarks that have been built in the burst-mode community to date. The *postoffice* [3], *cache-ctrl* [18], *diffeq* [23], *cd-player* [9], *pscsi* [24], *sscsi* [19], *xscsi* [22], *dram-ctrl* [19], and *barcode* [20] are all derived from real-life designs. The burst-mode controllers *ack-cdplayer, ack-fibonacci, ack-diffeq, ack-barcode, ack-gcd,* and *ack-factorial* are generated automatically from a procedural language description by the high-level synthesis framework ACK [12, 13, 14, 7, 6]. The other examples are classic burst-mode benchmarks from various publications. All benchmarks are run on a 333 MHz Ultrasparc-2 processor with 1 GB of physical memory and 840 MB of virtual memory. All benchmarks that finished ran completely in physical memory. Hence the results should show the true runtime potential of the respective minimizers.

As can be seen in Table 4, our logic minimization method is very fast for the vast majority of the benchmarks. As illustrated by the *ack-cdplayer-p1*, *ack-fibonacci*, and *ack-diffeq* benchmarks our algorithm is especially efficient when minimizing large and complex extended burst-mode controllers because of the compact way our method represents directed don't cares and level signals. For these benchmarks, our method is well over two orders of magnitude faster than the literal-exact tool HFMIN. While the IMPYMIN tool performs better runtime wise than HFMIN for large benchmarks, it can only produce cube exact covers. These covers have up to 25% more literals than corresponding minimal literal solutions. The ESPRESSO-HF minimizer performs well for all benchmarks in terms of runtime. Being a heuristic minimizer however, it produces covers with up to 19% more literals than literal exact solutions.

These observations underscore the importance of achieving fast literal exact minimization. Even when comparing against the heuristic algorithm of ESPRESSO-HF, ATACS is as fast or faster for all but one benchmark and, in addition, produces literal exact covers. In addition to simply being faster, our method can also perform literal exact minimization for designs where this has previously been impossible. For the *ack-cdplayer-p1* benchmark, HFMIN runs out of memory despite 1.84 GB of available memory while our minimizer ATACS completes in 80 seconds using less than 74 MB of memory. It is worth noting that the one benchmark, *cache-ctrl*, for which ATACS has a relatively long runtime, has an inordinate large number of required cubes. The large number of required cubes for this benchmark results in a diminishing return for ATACS otherwise effective divide and merge minimization strategy, explaining the longer runtime. Nevertheless, for this benchmark, ATACS is still on par with the implicit algorithm of IMPYMIN.

6 Conclusions

An efficient algorithm for single output literal-exact two-level logic minimization has been presented. The effectiveness of the presented logic minimizer can be mainly attributed to two factors. First, the signal concurrency properties of extended burst-mode controllers allows them to be expressed very efficiently in the form of compacted state graphs. Using compacted state graphs to represent extended burst-mode finite state machines, time spent in state graph exploration grows linearly, rather than exponentially, with the complexity (amount of signal concurrency and number of level signals) of the specification. The notion of compacted states is also exploited to significantly reduce the time spent in solving the binate and unate cover problems necessary to find hazard-free minimal solutions, as the size of the cover tables is significantly reduced. Second, the presented single-output minimization algorithm naturally divides the problem of finding a solution into smaller sub-problems of finding local unique solutions for each required cube separately, which are then merged into a final minimal solution for the entire output function. This divide and merge strategy has shown to perform well for most benchmarks and particularly where the number of required cubes is limited. Put together, these techniques form a literal-exact logic minimization approach that can perform several orders of magnitude faster than existing methods based on classical minimization approaches for large and complex controllers, making interactive and iterative design exploration possible.

Acknowledgement: The authors would like to thank Michael Theobald and Steve Nowick for their help with benchmarks and tools.

References

[1] P. Beerel and T.-Y. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 581–587. IEEE Computer Society Press, Nov. 1992.

[2] P. A. Beerel, C. J. Myers, and T. H.-Y. Meng. Covering conditions and algorithms for the synthesis of speed-independent circuits. *IEEE Transactions on Computer-Aided Design*, Mar. 1998.

[3] B. Coates, A. Davis, and K. Stevens. The Post Office experience: Designing a large asynchronous chip. *Integration, the VLSI journal*, 15(3):341–366, Oct. 1993.

[4] R. M. Fuhrer. *Sequential Optimization of Asynchronous and Synchronous Finite-State Machines*. PhD thesis, Department of Computer Science, Columbia University, 1999.

[5] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, B. Lin, and L. Plana. Minimalist: An environment for the synthesis, verification and testability of burst-mode asynchronous machines. Technical Report TR CUCS-020-99, Columbia University, NY, July 1999.

[6] H. Jacobson, E. Brunvand, G. Gopalakrishnan, and P. Kudva. High-level asynchronous system design using the ACK framework. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 93–103. IEEE Computer Society Press, Apr. 2000.

[7] H. Jacobson and G. Gopalakrishnan. Application-specific programmable control for high-performance asynchronous circuits. *Proceedings of the IEEE*, 87(2):319–331, Feb. 1999.

[8] H. Jacobson, C. J. Myers, and G. Gopalakrishnan. Achieving fast and exact hazard-free logic minimization of extended burst-mode gC finite state machines. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, Nov. 2000.

[9] J. Kessels, K. van Berkel, R. Burgess, M. Roncken, and F. Schalij. An error decoder for the compact disc player as an example of VLSI programming. Technical report, Philips Research Laboratories, Eindhoven, The Netherlands, 1992.

[10] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. Analysis and identification of speed-independent circuits on an event model. *Formal Methods in System Design*, 4(1):33–75, 1994.

[11] A. Kondratyev, M. Kishinevsky, and A. Yakovlev. Hazard-free implementation of speed-independent circuits. *IEEE Transactions on Computer-Aided Design*, 17(9):749–771, Sept. 1998.

[12] P. Kudva. *Synthesis of Asynchronous Systems Targeting Finite State Machines*. PhD thesis, Computer Science Department, University of Utah, 1995.

[13] P. Kudva, G. Gopalakrishnan, and H. Jacobson. A technique for synthesizing distributed burst-mode circuits. In *Proc. ACM/IEEE Design Automation Conference*, 1996.

[14] P. Kudva, G. Gopalakrishnan, H. Jacobson, and S. M. Nowick. Synthesis of hazard-free customized CMOS complex-gate networks under multiple-input changes. In *Proc. ACM/IEEE Design Automation Conference*, 1996.

[15] A. Marshall, B. Coates, and P. Siegel. Designing an asynchronous communications chip. *IEEE Design & Test of Computers*, 11(2):8–21, 1994.

[16] C. J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Dept. of Elec. Eng., Stanford University, Oct. 1995.

[17] S. M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers*. PhD thesis, Stanford University, Department of Computer Science, 1993.

[18] S. M. Nowick, M. E. Dean, D. L. Dill, and M. Horowitz. The design of a high-performance cache controller: a case study in asynchronous synthesis. *Integration, the VLSI journal*, 15(3):241–262, Oct. 1993.

[19] S. M. Nowick, K. Y. Yun, and D. L. Dill. Practical asynchronous controller design. In *Proc. International Conf. Computer Design (ICCD)*, pages 341–345. IEEE Computer Society Press, Oct. 1992.

[20] P. R. Panda and N. Dutt. 1995 high level synthesis design repository. Technical Report 95-04, University of California, Irvine, U.S.A., 1995.

[21] M. Theobald and S. M. Nowick. Fast heuristic and exact algorithms for two-level hazard-free logic minimization. *IEEE Transactions on Computer-Aided Design*, 17(11):1130–1147, Nov. 1998.

[22] K. Y. Yun. *Synthesis of Asynchronous Controllers for Heterogeneous Systems*. PhD thesis, Stanford University, Aug. 1994.

[23] K. Y. Yun, P. A. Beerel, V. Vakilotojar, A. E. Dooply, and J. Arceo. The design and verification of a high-performance low-control-overhead asynchronous differential equation solver. *IEEE Transactions on VLSI Systems*, 6(4):643–655, Dec. 1998.

[24] K. Y. Yun and D. L. Dill. Automatic synthesis of 3D asynchronous state machines. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 576–580. IEEE Computer Society Press, Nov. 1992.

[25] K. Y. Yun and D. L. Dill. Automatic synthesis of extended burst-mode circuits: Part I (specification and hazard-free implementation). *IEEE Transactions on Computer-Aided Design*, 18(2):101–117, Feb. 1999.